# Module de suivi de trajectoires en temps réel pour un robot omnidirectionnel non-holonome

Francis BISSON
Francis.Bisson@USherbrooke.ca

Simon CHAMBERLAND*
Simon.Chamberland@USherbrooke.ca

Laboratoire PLANIART
Département d'informatique
Université de Sherbrooke
Sherbrooke (QC), Canada, J1K 2R1
http://planiart.usherbrooke.ca/

**- Rapport de projet pour le cours IFT 729[1] -**

**Résumé**

Ce rapport rend compte du travail accompli pour le projet réalisé dans le cadre du cours IFT 729. Nous avons implémenté un module de suivi de trajectoires, respectant des contraintes de temps réel, pour un robot omnidirectionnel non-holonome. Nous avons dû ajouter des fonctionnalités à la plateforme de simulation OOPSMP afin de permettre l'exécution des trajectoires de façon non-déterministe. Le module du suivi de trajectoires peut aussi être utilisé pour d'autres types de robots et d'objets mobiles. Nous avons donc aussi implémenté un générateur de trajectoires pour une voiture simple afin d'y tester notre module de suivi de trajectoires. Les expérimentations que nous avons effectuées et les résultats qui en découlent démontrent qu'il est possible d'avoir un module de suivi de trajectoires efficace, même s'il est soumis à contraintes de temps réel.

**Mots-clés :** Planification de trajectoires, suivi de trajectoires, robotique, contraintes holonomes.

*Auteur à contacter.

[1] Conception de systèmes temps réel.

# Table des matières

**1 Introduction** **4**
1.1 Mise en contexte . . . . . 4
1.2 Description du projet . . . . . 5
1.3 Objectifs visés . . . . . 5

**2 Implémentation** **6**
2.1 Architecture haut niveau . . . . . 6
2.2 Exécuteur de plans . . . . . 7
2.3 Simulateur non-déterministe . . . . . 9
2.4 Module de suivi de trajectoires . . . . . 10
2.5 Fichiers modifiés . . . . . 11

**3 Contraintes de temps réel** **13**
3.1 Borne supérieure sur le temps de simulation . . . . . 13
3.2 Temps de calcul alloué pour le suivi de trajectoires . . . . . 13

**4 Expérimentations et analyse des résultats** **14**
4.1 Tests empiriques . . . . . 14
4.2 Généralisation du module de suivi de trajectoires . . . . . 15

**5 Conclusion** **17**

**A Mode d'emploi** **18**
A.1 Prérequis . . . . . 18
A.2 Installation . . . . . 18
A.3 Compilation du projet . . . . . 18
A.4 *Controller Chooser* . . . . . 20
A.5 Exécution . . . . . 21
A.5.1 Planification d'une trajectoire . . . . . 21
A.5.2 Exécution d'une trajectoire . . . . . 22
A.6 Désinstallation . . . . . 22

**B Génération de trajectoires pour une voiture simple** **23**
B.1 Hypothèses . . . . . 23
B.2 Modèle mathématique . . . . . 23
B.3 Génération d'une trajectoire . . . . . 24

**Bibliographie** **26**

# Liste des tableaux

2.1 Fichiers modifiés dans OOPSMP . . . . . 12

# Liste des figures

1.1 Le robot omnidirectionnel AZIMUT-3 . . . . . 4
2.1 Diagramme UML des classes `PlanExecutor`, `Simulator` et `FeedbackPlanner` . . . 6
2.2 Diagramme de séquence des modèles d'exécution et de suivi de trajectoires . . . . 8
2.3 Champ de potentiel pour un environnement 2D simple . . . . . 10
2.4 Points de cheminement sur une trajectoire . . . . . 11
4.1 Environnements de simulation . . . . . 14
4.2 Obstacle entre le robot et le point de cheminement . . . . . 15
4.3 Voiture coincée entre deux obstacles . . . . . 16
A.1 Configuration de CMake . . . . . 19
A.2 Interface du *Controller Chooser* . . . . . 20
A.3 Interface graphique de OOPSMP . . . . . 21
B.1 Modèle mathématique d'une voiture simple . . . . . 23

# 1 Introduction

## 1.1 Mise en contexte

La planification de trajectoires est un important problème de recherche en intelligence artificielle qui consiste à calculer une trajectoire géométrique d'un état initial à une configuration finale, pour un objet mobile donné [1]. Les trajectoires générées doivent aussi tenir compte de l'environnement (e.g., obstacles, autres objets en mouvement, etc.) et des contraintes cinématiques de l'objet mobile. Puisque l'espace d'état est, dans la plupart des cas, continu plutôt que discret, des algorithmes d'échantillonnage aléatoire sont souvent utilisés. OOPSMP est un logiciel extensible de planification de trajectoires implémentant plusieurs tels algorithmes [4].

Le laboratoire IntRoLab[2] de la Faculté de génie de l'Université de Sherbrooke a conçu plusieurs robots mobiles, dont un robot omnidirectionnel non-holonome nommé AZIMUT-3 (figure 1.1). Les modules nécessaires à la simulation correcte de la plateforme AZIMUT-3 dans OOPSMP ont déjà été implémentés par Daniel Castonguay et Simon Chamberland dans le cadre d'un projet effectué en collaboration avec le laboratoire PLANIART lors de la session d'hiver 2009[3].

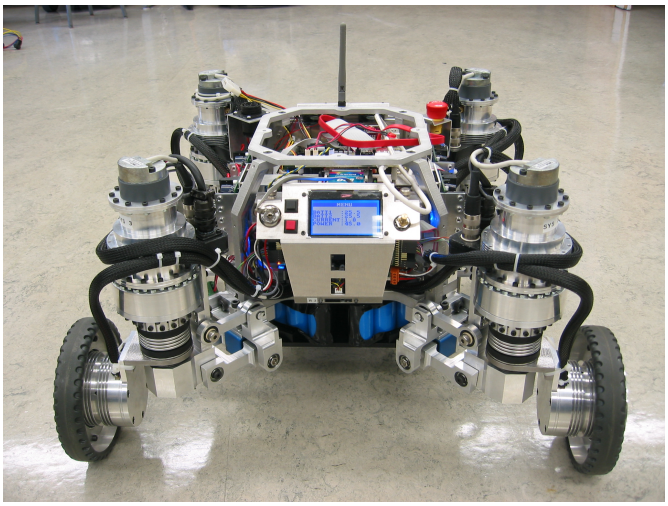


FIG. 1.1 – Le robot omnidirectionnel AZIMUT-3.

Bien qu'il soit maintenant possible de planifier des trajectoires pour le robot AZIMUT-3 dans OOPSMP, il serait souhaitable d'implémenter de nouvelles fonctionnalités dans le simulateur, permettant de mieux refléter la réalité. De plus, certains ajouts pourraient aussi bénéficier le planificateur de trajectoires intégré au robot lui-même.

[2] `http://www.gel.usherbrooke.ca/laborius/`
[3] `https://planiart.usherbrooke.ca/redmine/projects/show/motionplan-azimut3`

## 1.2 Description du projet

En robotique mobile, le suivi de trajectoires est un problème qui concerne les imprédictibilités du monde réel [3]. En effet, lorsqu'une trajectoire est planifiée pour un robot autonome se déplaçant dans son environnement, il est fort probable que l'environnement modélisé pour effectuer la planification et la simulation ne corresponde pas exactement à la réalité. L'environnement peut n'être que partiellement connu *a priori*, être trop complexe à modéliser parfaitement, ou avoir changé entre le moment de la planification et de l'exécution de la trajectoire (e.g., un astromobile sur la planète Mars). De plus, il est tout à fait possible que les capteurs et les actionneurs du robot soient imprécis, ce que ajoute de l'incertitude à ses déplacements. Un module de suivi de trajectoires permettant au robot de réajuster son état afin de rester le plus près possible de sa trajectoire calculée s'avère souvent non seulement souhaitable, mais nécessaire.

Dans le cadre du cours IFT 729, nous avons donc voulu implémenter un module de suivi de trajectoires soumis à des contraintes de temps réel (section 3), qui pourrait être utilisé par le robot AZIMUT-3. Idéalement, le module devrait être suffisamment générique pour fonctionner avec d'autres types de corps mobiles. De façon abstraite, le module de suivi de trajectoires devrait fournir une fonction $f : X \rightarrow U$ (où $X$ est l'espace d'états et $U$ est l'espace des contrôles) qui, étant donné un état $x \in X$, donne un contrôle (i.e., une action) $u \in U$ à appliquer pour réintégrer la trajectoire initiale. Une complexité algorithmique $O(1)$ serait souhaitable pour cette fonction.

## 1.3 Objectifs visés

Voici une liste des principaux problèmes adressés et objectifs visés par ce projet :

1. Concevoir un simulateur non-déterministe à l'intérieur de OOPSMP ;
2. Expérimenter avec divers modèles d'exécution non-déterministe permettant de représenter plus ou moins fidèlement le monde réel ;
3. Implémenter un module de suivi de trajectoires pour le robot AZIMUT-3 ;
4. Assurer des contraintes de temps réel pour effectuer le suivi de trajectoires ;
5. Généraliser le module de suivi de trajectoires pour le rendre fonctionnel avec d'autres types d'objets mobiles.

Comme nous l'avons déjà indiqué dans le Livrable 01, il nous a été impossible d'implanter notre projet sur le robot AZIMUT-3 ; nous avons dû rester en environnement de simulation. Les objectifs 1 et 2 ont également été discutés dans le Livrable 01 mais seront traités sommairement dans le présent rapport en guise de rappel.

# 2 Implémentation

## 2.1 Architecture haut niveau

Puisque nous travaillons dans un environnement de simulation et non directement sur le robot lui-même, nous avions besoin d'un moyen d'introduire de l'incertitude sur les actions exécutées par le robot simulé, afin de représenter fidèlement la réalité. En effet, un robot réel agira de façon plus ou moins incertaine, que ce soit à cause de ses capteurs bruités, de ses actionneurs imprécis ou de l'environnement dans lequel la planification a été effectuée qui diffère de l'environnement réel.

Afin de permettre l'introduction d'incertitude dans l'exécution des trajectoires par le robot, nous avons ajouté un module de simulation générique composé d'une triade de classes de base : un exécuteur de plans (`PlanExecutor`), un simulateur (`Simulator`) et un module de suivi de trajectoires (`FeedbackPlanner`). Un diagramme UML de ces classes est illustré dans la figure 2.1.

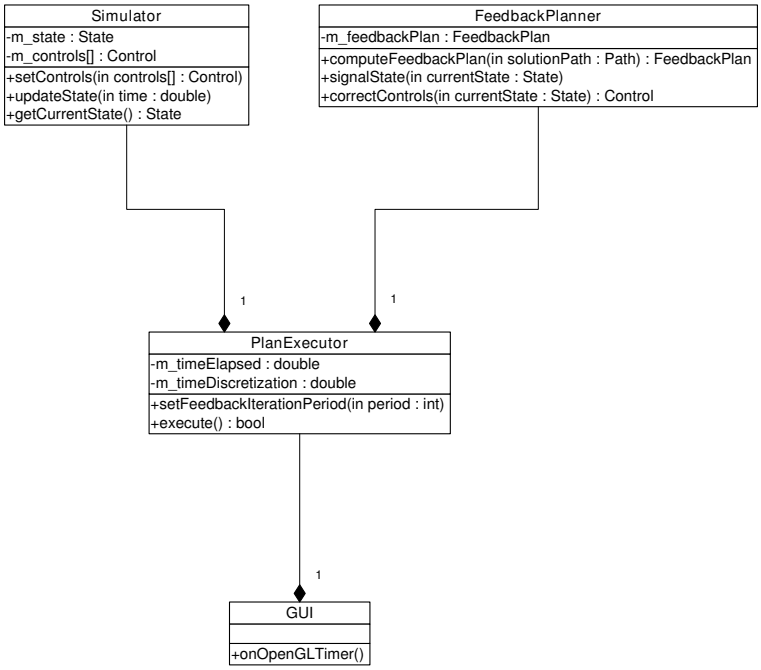



FIG. 2.1 – Diagramme UML des classes `PlanExecutor`, `Simulator` et `FeedbackPlanner`.

Le rôle de chacune de ces classes est bien précis, et l'architecture pour laquelle nous avons opté permet de bien respecter leur encapsulation. De plus, cette architecture permet aussi d'unifier les cas déterministe et non-déterministe de façon élégante.

Parallèlement à cette architecture, des obstacles dynamiques ont été introduits dans l'environnement. Ceux-ci ont été implémentés comme des voitures simples se déplaçant aléatoirement dans l'environnement.

## 2.2 Exécuteur de plans

La classe `PlanExecutor` est la seule qui est utilisée directement par le code client. Elle se sert des deux autres pour demander au robot d'exécuter un plan (i.e., une trajectoire). Le programme principal appelle itérativement la méthode `execute()` de l'exécuteur de plans avec un certain pas de discrétisation paramétrable, qui ne correspond pas nécessairement au temps écoulé dans la réalité. De plus, dans le cas non-déterministe, cette méthode ne s'exécute présentement pas en temps constant, sauf pour la classe `NonDeterministicRealtimePlanExecutor` qui, comme son nom l'indique, impose une contrainte de temps réel sur le temps alloué pour l'exécution d'une trajectoire.

Nous avons étudié trois modèles d'exécution et de suivi de trajectoires par le robot. Ces modèles sont décrits dans le diagramme de séquence de la figure 2.2.

Le premier modèle est constitué d'appels synchrones entre le module de suivi de trajectoires et le simulateur. Le problème est qu'un temps de calcul non-constant est nécessaire pour trouver le nouvel état du robot, étant donné un contrôle (i.e., une action) à appliquer. En effet, des méthodes d'intégration numérique sont parfois nécessaire (voir la section B.2 pour un contre-exemple) pour mettre à jour son état. Il faut de plus vérifier les collisions avec les obstacles qui se trouvent dans l'environnement. Il n'est donc pas toujours possible de savoir en temps constant à quelle position se trouvera le robot après un certain délai $\Delta t$. Ceci est problématique dans le sens où, la réalité étant évidemment en temps réel, le robot peut lire ses capteurs et connaître en un temps constant sa position, l'orientation de ses roues, etc. Il n'y a pas d'intégration coûteuse à calculer, ni de test de collision avec l'environnement à effectuer.

Ceci nous amène donc au deuxième modèle d'exécution, où le simulateur tient compte du temps de calcul requis par le module de suivi de trajectoires. Ce modèle permet donc de mieux représenter la réalité puisque le robot continue d'appliquer le contrôle précédent (e.g., avancer) pendant que le module de suivi de trajectoires calcule une nouvelle action à exécuter pour rejoindre la trajectoire initiale. Ce modèle d'exécution utilise cependant toujours des appels synchrones entre le module de suivi de trajectoires et le simulateur. Dans l'implémentation, l'exécuteur de plans demande au module de suivi de trajectoires de calculer, pendant un certain temps qui lui est alloué, un contrôle à appliquer permettant au robot de réintégrer sa trajectoire originale. Une fois le calcul effectué, le simulateur applique le contrôle précédent pendant le temps utilisé par le module de suivi de trajectoires, de façon à simuler l'exécution de cette même action pendant que le planificateur

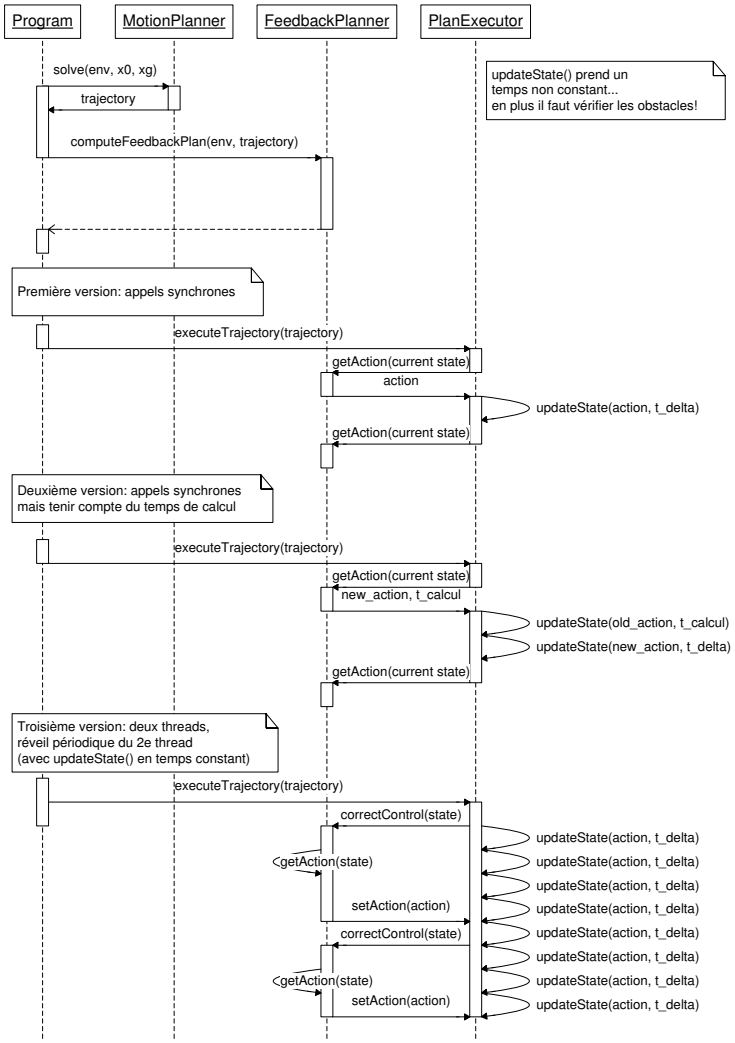



FIG. 2.2 – Diagramme de séquence des modèles d'exécution et de suivi de trajectoires.

effectuait son travail. Le simulateur peut ensuite appliquer le nouveau contrôle calculé par le module de suivi de trajectoires.

Le troisième et dernier modèle utilise des appels asynchrones entre le module de suivi de trajectoires et le simulateur, de façon à être aussi fidèle à la réalité que possible. Ce modèle nécessite la création de deux fils d'exécution parallèles : un pour le module de suivi de trajectoires et un autre pour le simulateur. Le premier reçoit périodiquement, de la part du simulateur, un message lui demandant de calculer un contrôle à appliquer pour rejoindre la trajectoire initiale, étant donné l'état actuel du robot. Celui-ci ne fait qu'appliquer continuellement une action, qui est modifiée de temps à autre par le module de suivi de trajectoires. En plus d'être fidèle au monde réel, le modèle serait relativement aisé à implanter dans le robot AZIMUT-3 ; le simulateur servant à simuler la

réalité, il n'aurait qu'à être remplacé par les modules du robot servant à effectuer des lectures sur les capteurs. De la synchronisation doit toutefois être ajouté pour assurer l'intégrité des données. En particuler, les nouvelles actions générées par le module de suivi de trajectoire ne doivent pas corrompre celles en cours de "'consommation"' par le simulateur.

Afin de supporter ce modèle, nous avons dû apporter quelques modifications aux bibliothèques que nous utilisons dans le projet. En effet, les bibliothèques *Kinetics* (développée par des membres du laboratoire IntRoLab) et PQP (*Proximity Query Package*) n'étaient pas *thread-safe*.

La bibliothèque *Kinetics* utilisait deux bibliothèques qui causaient des problèmes. Pour la première, CLAPACK (un ensemble d'outils d'algèbre linéaire), nous n'avons eu qu'à utiliser la version la plus récente, distribuée gratuitement sur Internet[4] sous une licence libre. La deuxième, ANN[5] (*Approximate Nearest Neighbor*), utilisait des variables globales pour éviter des listes trop volumineuses de paramètres formels dans les fonctions de l'interface. Nous avons donc créé une structure de données contenant l'ensemble des paramètres à passer entre les fonctions.

En ce qui concerne la bibliothèque PQP, nous nous sommes inspirés du concept de preuve qui a été vu en classe. En effet, cette bibliothèque est utilisée exclusivement à travers l'appel `StateSpace::applyState(x)`, qui est suivi typiquement d'une vérification de collisions ou d'un affichage à l'écran. Introduire un auto-verrou dans la méthode `applyState(x)` est insuffisant, puisque l'état en question est utilisé plus loin et nous souhaitons qu'il ne soit pas modifié. La solution que nous avons retenue consiste à ajouter un paramètre `Lock` à l'appel de `applyState(x)`, qui permet de s'assurer que l'appelant a acquis le verrou *a priori*. Le verrou est relâché dans le destructeur de `Lock`; nous comptons donc sur l'appelant pour réduire la portée de l'objet `Lock` au maximum.

## 2.3 Simulateur non-déterministe

La classe abstraite `Simulator`, ou plutôt sa sous-classe concrète non-déterministe, est celle qui introduit du bruit pseudo-aléatoire au contrôle du robot, qui est l'action (e.g., tourner, accélérer, etc.) que le robot doit appliquer pour une certaine durée donnée. Dans le cas déterministe, une autre sous-classe concrète ne fait que transmettre le contrôle fourni par le planificateur au robot, sans lui ajouter de bruit. Pour l'instant, le seul simulateur non-déterministe que nous avons implémenté applique du bruit gaussien sur le contrôle du robot, avec un écart-type paramétrable (voir la section 4). Il serait bien sûr possible d'ajouter d'autres types de simulateurs pseudo-aléatoires, mais nous n'avions pas accès à des données de simulations réelles à partir desquelles établir une distribution probabiliste.

[4] `http://www.netlib.org/clapack/`
[5] `http://www.cs.umd.edu/~mount/ANN/`

## 2.4 Module de suivi de trajectoires

La classe `FeedbackPlanner` est le cœur de notre projet. Il s'agit du module servant à rectifier la trajectoire du robot qui s'en serait égaré suite à une exécution non-déterministe.

Pour concevoir ce module, nous voulions utiliser une technique inspirée des champs de potentiel [2]. La figure 2.3 illustre un exemple de champ de potentiel dans un environnement bidimensionnel simple. Les cases noires sont des obstacles et les cases grises sont des positions à éviter. Le triangle vert indique la position initiale du robot et le cercle rouge donne l'état final à atteindre. Les flèches indiquent au robot dans quelle direction il devrait se déplacer dans tous les états possibles afin de rejoindre sa trajectoire initiale.

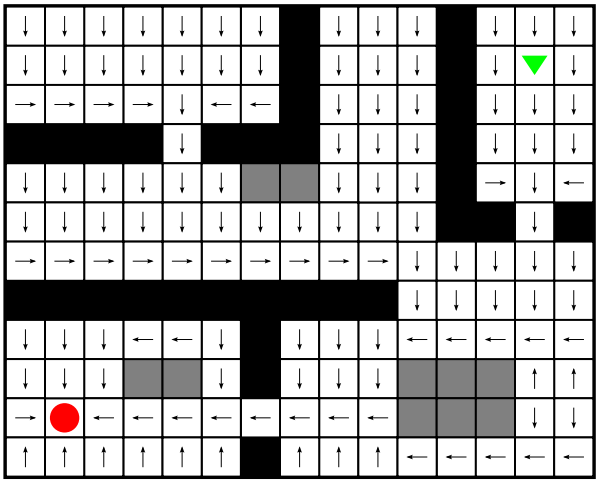


FIG. 2.3 – Champ de potentiel pour un environnement 2D simple.

Cette technique est appropriée pour un système soumis à des contraintes de temps réel puisqu'elle permet d'obtenir, dans un temps constant, l'action que le robot doit exécuter pour rejoindre sa trajectoire initiale s'il s'en est égaré. Cependant, la planification de trajectoires en robotique mobile opère souvent sur un environnement continu et non discret ; c'est le cas ici. L'espace d'état a donc une dimension infinie. Associer une action à chaque état nécessite alors une discrétisation de l'environnement, à des niveaux de granularité plus ou moins fins. De plus, le calcul du champ de potentiel requiert bien entendu un temps de calcul considérable.

Pour ces raisons et à cause d'un manque de temps, nous avons opté pour une autre technique de calcul du contrôle que le robot doit appliquer pour rejoindre sa trajectoire initiale. Nous avons conçu un algorithme *ad hoc* basé sur les points de cheminement (*waypoints*). La figure 2.4 donne un exemple d'une telle méthode dans un environnement 2D simple. Le triangle vert indique la position initiale du robot et le cercle rouge donne l'état final à atteindre. Les étoiles magenta sont les points de cheminements situés sur la trajectoire (continue) du robot, illustrée par la ligne bleue.

Cette technique consiste à discrétiser la trajectoire solution de façon à en extraire régulièrement (e.g.., à toutes les $x$ secondes) des états qui serviront de points de cheminement. Afin de se rendre à destination, le robot devra donc passer par l'ensemble de ces points de cheminement, de façon séquentielle.

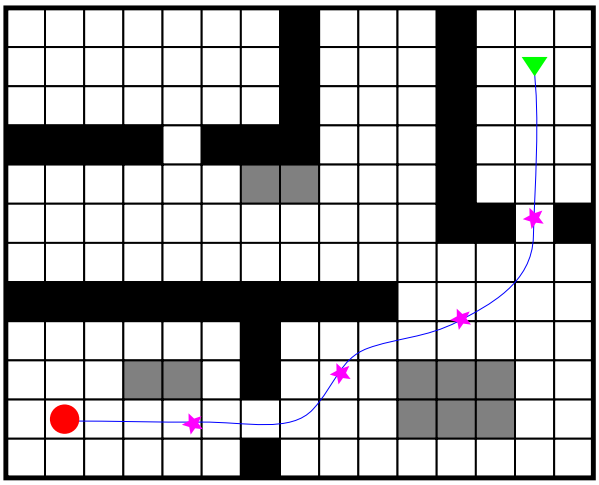


FIG. 2.4 – Points de cheminement sur une trajectoire.

Dans l'interface graphique, le point de cheminement courant est représenté avec la couleur magenta. Le module de suivi de trajectoires tente constamment de générer une action lui permettant de se rapprocher le plus possible de ce point. La génération de cette action est confiée au générateur de trajectoires. Selon l'implémentation de celui-ci, la génération d'action peut prendre un temps constant, ou dépendre des paramètres de la fonction (état initial et état final désiré). Nous pouvons toutefois certifier qu'une borne maximale sur le temps d'exécution existe, puisque les différents générateurs de trajectoires utilisés ne font normalement pas usage de boucles de contrôle (`for`, `while`, etc.).

Si l'évitement d'obstacles est activé (par l'option *Avoid obstacles* dans le *Controller Chooser*), le générateur de trajectoires calculera des trajectoires n'entrant pas en contact avec les obstacles statiques et dynamiques. Cependant, l'incertitude introduite par le simulateur (`GaussianSimulator`), l'incertitude sur les déplacements des obstacles dynamiques ainsi que le délai de correction des trajectoires font en sorte que des collisions peuvent tout de même se produire.

## 2.5 Fichiers modifiés

Le tableau 2.1 liste les principaux endroits où des fichiers ont été modifiés dans OOPSMP. D'autres modifications moins importantes ont été effectuées un peu partout dans le code afin

d'adapter les interfaces et d'ajouter des fonctionnalités requises par les nouveaux modules que nous avons implémentés. Les fichiers que nous avons créé devraient généralement contenir une entête en commantaire faisant mention du laboratoire PLANIART.

| **Répertoire** | **Raison** |
|---|---|
| `Selector\*` | Modifications aux *Controller Chooser* |
| `src\PlanExecution\*` | Simulation non-déterministe et suivi de trajectoires |
| `src\Core\Path\Plugin\` | Voiture simple (`OOPSMPKCarPathGenerator.H`, `OOPSMPKCarPathGenerator.cpp`, `OOPSMPKCarStateSpace.H` et `OOPSMPKCarStateSpace.cpp`) |

Tab. 2.1 – Fichiers modifiés dans OOPSMP.

# 3 Contraintes de temps réel

## 3.1 Borne supérieure sur le temps de simulation

La première contrainte de temps réel à imposer pour notre projet concerne le temps maximal alloué à l'exécuteur de plans pour effectuer son travail. Dans l'optique de modéliser fidèlement la réalité, l'exécuteur de plans doit réaliser l'intégration numérique (i.e., calculer le prochain état du robot après un certain temps) et la vérification des collisions avec des obstacles de l'environnement en temps constant.

Pour ce faire, nous avons établi une borne supérieure sur le temps de calcul requis pour faire la simulation et nous utilisons ce temps maximal pour chaque tranche de temps allouée au simulateur. Bien qu'il s'agit d'une contrainte de temps réel stricte (puisqu'un non-respect de celle-ci ne permet pas de bien simuler le monde réel), imposer ce genre de contrainte est quasi impossible sur un système d'exploitation grand public (qui ne permet donc pas un contrôle précis sur les tranches de temps des processus en exécution). Nous forçons tout de même l'exécuteur de plans à ne pas dépasser son temps alloué à l'aide d'une minuterie *ad hoc*, sans quoi un message d'erreur est affiché dans la console. Aucune allocation dynamique de mémoire n'est faite dans les méthodes de la classe `NonDeterministicRealtimePlanExecutor` ; la mémoire est pré-allouée et réutilisée pour la durée de vie de l'objet. Ceci, combiné à des opérations s'effectuant en temps constant, garantit le respect de la première contrainte de temps réel.

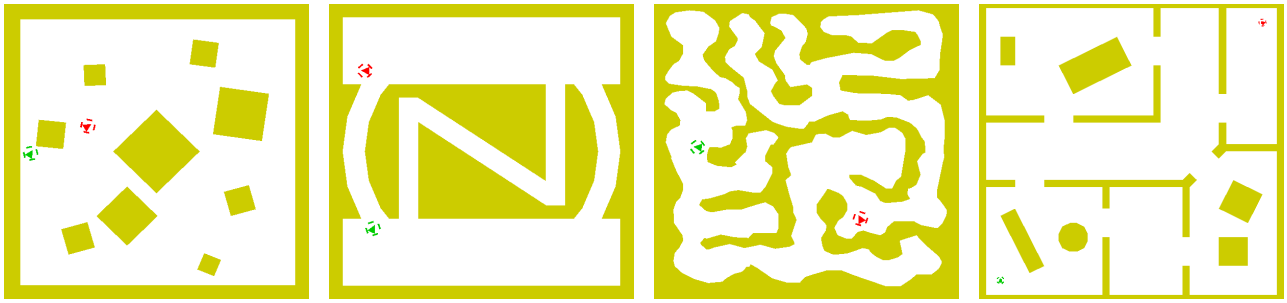
## 3.2 Temps de calcul alloué pour le suivi de trajectoires

La deuxième contrainte de temps réel concerne plutôt le temps alloué au module de suivi de trajectoires pour déterminer le contrôle que le robot doit appliquer, en tenant compte du facteur d'échelle du temps de la simulation (e.g., 1 seconde de temps simulé correspond à 3 secondes de temps réel). Étant donné que nous utilisons le 3$^{e}$ modèle d'éxécution (section 2.2), le seul effet du module de suivi de trajectoires mettant trop de temps à fournir un nouveau contrôle est que le robot continuera d'appliquer l'ancienne action. Cela peut bien entendu entraîner une collision avec un obstacle de l'environnement, conséquence très néfaste dans la vie réelle. Il s'agit donc aussi d'une contrainte de temps réelle stricte, même si, dans l'implémentation, il n'y a pas de conséquence fatale pour l'exécution de l'application si le temps permis est écoulé.

# 4 Expérimentations et analyse des résultats

## 4.1 Tests empiriques

Nous avons effectué plusieurs expérimentations dans les environnements de simulation illustrés dans la figure 4.1. Notez que l'environnement de la figure 4.1(d) est deux fois plus grand que les autres, d'où la taille réduite des polygones vert et rouge représentant le robot dans sa configuration initiale et finale, respectivement. Nous avons utilisé le modèle du robot omnidirectionnel non-holonome AZIMUT-3 pour effectuer nos expérimentations.



(a) Environnement 1. (b) Environnement 2. (c) Environnement 3. (d) Environnement 4.

FIG. 4.1 – Environnements de simulation.

Un manque de temps nous a empêché de fournir des tableaux et des graphiques de résultats empiriques pour nos expérimentations. Néanmoins, nous avons effectué plusieurs tests nous amenant à conclure que le module de suivi de trajectoires fonctionne bien et que les contraintes de temps réel énoncées dans la section 3 sont bien respectées. En effet, puisque le robot AZIMUT-3 entre relativement rarement en collision avec des obstacles statiques de son environnement (même si celui-ci est très complexe), cela signifie que le module de suivi de trajectoires réussi à réagir suffisamment vite selon les contraintes imposées pour empêcher une collision. Les collisions avec les obstacles statiques surviennent, la plupart du temps, lorsqu'un obstacles est placé entre le robot et son prochain point de cheminement. Dans cette situation (illustrée dans la figure 4.2), le robot peine à faire un détour pour éviter l'obstacles et rejoindre sa destination. Cependant, les obstacles dynamiques, de par leurs comportements aléatoires imprévisibles, causent plus souvent des collisions avec le robot. Il est important de noter que notre algorithme de suivi de trajectoires (ni celui de planification de trajectoires, d'ailleurs) ne tient pas compte des obstacles dynamiques, ce qui explique les collisions qui ne sont pas évitées.

Les messages d'erreur indiquant que l'exécuteur de plans a dépassé sa tranche de temps n'apparaîssent généralement que lorsque l'application a été compilée en mode *Debug*, ou lorsque plusieurs processus s'exécutent en même temps sur l'ordinateur hôte et que celui d'OOPSMP se

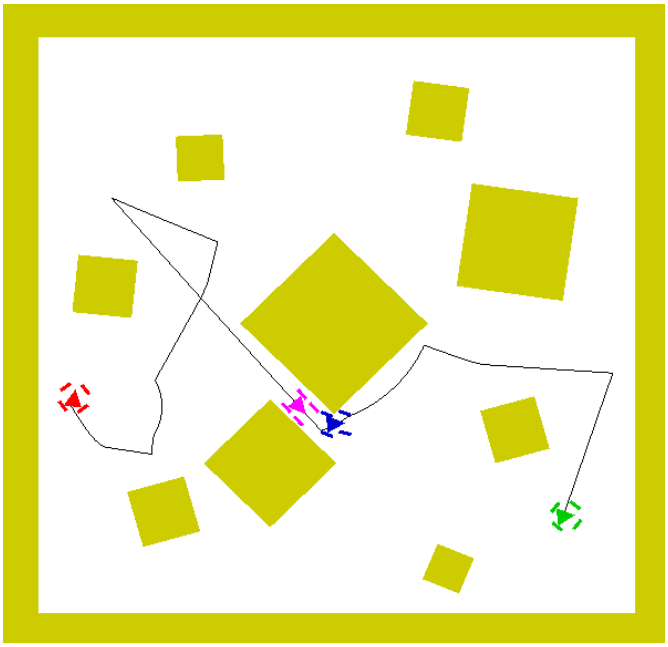


FIG. 4.2 – Obstacle entre la roue arrière droite du robot et le point de cheminement.

fait interrompre. Puisque nous utilisons un système d'exploitation grand public (Windows XP ou Vista), il ne nous est pas possible d'empêcher cette situation. De plus, nos tests ont démontré que la taille et la complexité de l'environnement n'influencent pas le temps pris par l'exécuteur de plans pour effectuer son travail, ce qui confirme que les opérations s'exécutent en temps constant.

## 4.2 Généralisation du module de suivi de trajectoires

Afin de valider la bonne généralisation du module de suivi de trajectoires réalisé dans le cadre de ce projet, nous avons implémenté un générateur de trajectoires pour une voiture simple. Les détails d'implémentation ainsi que les raisonnements mathématiques derrière un tel générateur de trajectoires sont donnés dans l'annexe B.

Nos expérimentations avec le modèle de la voiture simple ont démontré que le module de suivi de trajectoires basé sur les points de cheminement fonctionnait effectivement avec d'autres types de corps mobiles que celui pour lequel il a été conçu en premier lieu. Cependant, nous avons remarqué que la voiture entrait en collision avec des obstacles dans l'environnement beaucoup plus souvent que le robot AZIMUT-3. Cela s'explique sans doute par le fait que le robot AZIMUT-3 peut se déplacer dans toutes les directions, alors que la voiture ne peux qu'avancer et reculer soit en ligne droite ou sur un arc de cercle. Certaines situations, par exemple celle illustrée dans la figure 4.3, font

en sorte que la voiture peut mettre beaucoup de temps (et entrer souvent en collision avec un ou des obstacles) avant de réussir à se rendre à sa destination. En effet, la voiture ne pouvant se déplacer latéralement, elle doit continuellement reculer et avancer en espérant réussir à se réorienter dans la direction du point de cheminement suivant, de façon analogue à un stationnement en parallèle avec une vraie voiture.

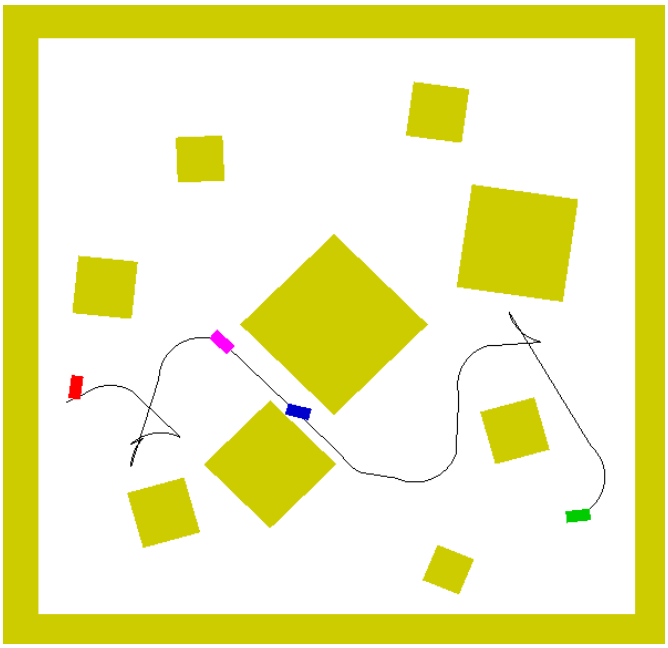


FIG. 4.3 – Voiture coincée entre deux obstacles.

# 5 Conclusion

Avant de pouvoir implémenter l'algorithme sur le robot réel, nous devrons améliorer le module de suivi de trajectoires afin qu'il fonctionne correctement dans la plupart des cas problématiques que nous avons identifiés. En particulier, si le robot ne peut plus avancer parce qu'il fait face à un mur, ou si un point de cheminement n'est plus accessible à cause de la présence d'obstacles dynamiques, il pourrait être de mise d'abandonner les points de cheminement en faveur d'une replanification complète jusqu'à l'état final. Certains algorithmes, comme D* [5], permettent d'utiliser certaines informations déjà calculées afin d'accélérer le processus de replanification.

De plus, nous devrons nous assurer que les bornes de temps du module de suivi de trajectoires sont aussi respectés sur le robot, car celui-ci possède de l'équipement (*hardware*) moins performant que les ordinateurs sur lesquels nous travaillions. Il sera aussi nécessaire de vérifier l'allocation de temps au module de suivi de trajectoires par rapport aux autres modules afin d'ajuster le temps alloué à l'optimisation de la trajectoire corrigée.

# A Mode d'emploi

## A.1 Prérequis

Voici une liste des prérequis (en ordre alphabétique) pour pouvoir compiler, installer et exécuter le projet.

- Boost[6] 1.42
- CMake[7] 2.8
- grid.bin
- *Kinetics*
- OpenGL, GLU et GLUT
- Visual C++ .NET

Tous ces logiciels (sauf Visual Studio .NET) et ces bibliothèques sont fournis avec le code source du projet. Notez que nous n'avons testé l'application que sous Windows (XP et Vista) seulement, bien qu'il soit possible qu'elle fonctionne sous d'autres systèmes d'exploitation.

## A.2 Installation

L'installation du projet ne consiste qu'en quelques étapes très simples. Il faut d'abord installer CMake et Visual Studio .NET selon la procédure standard, ainsi que les bibliothèques Boost, *Kinetics*, OpenGL, GLU et GLUT. Il faut s'assurer que ces bibliothèques sont connues par le compilateur de Visual Studio .NET. Il faut ensuite extraire l'archive contenant le code source du projet dans un répertoire au choix. Finalement, le répertoire `src\sparsehash-1.4\src` (situé dans l'arborescence du code source du projet) doit aussi être accessible dans les dossiers d'inclusion du compilateur.

## A.3 Compilation du projet

OOPSMP utilise CMake comme moteur de production. Il faut donc configurer CMake de façon à ce que toutes les bibliothèques nécessaires soient trouvées. La figure A.1 donne l'exemple à suivre pour effectuer une bonne configuration de CMake (remplacer `C:/udes/ift729/oopsmp/`

[6] `http://www.boost.org/`
[7] `http://www.cmake.org/`

par le chemin absolu vers le répertoire où les sources du projet ont été extraites).

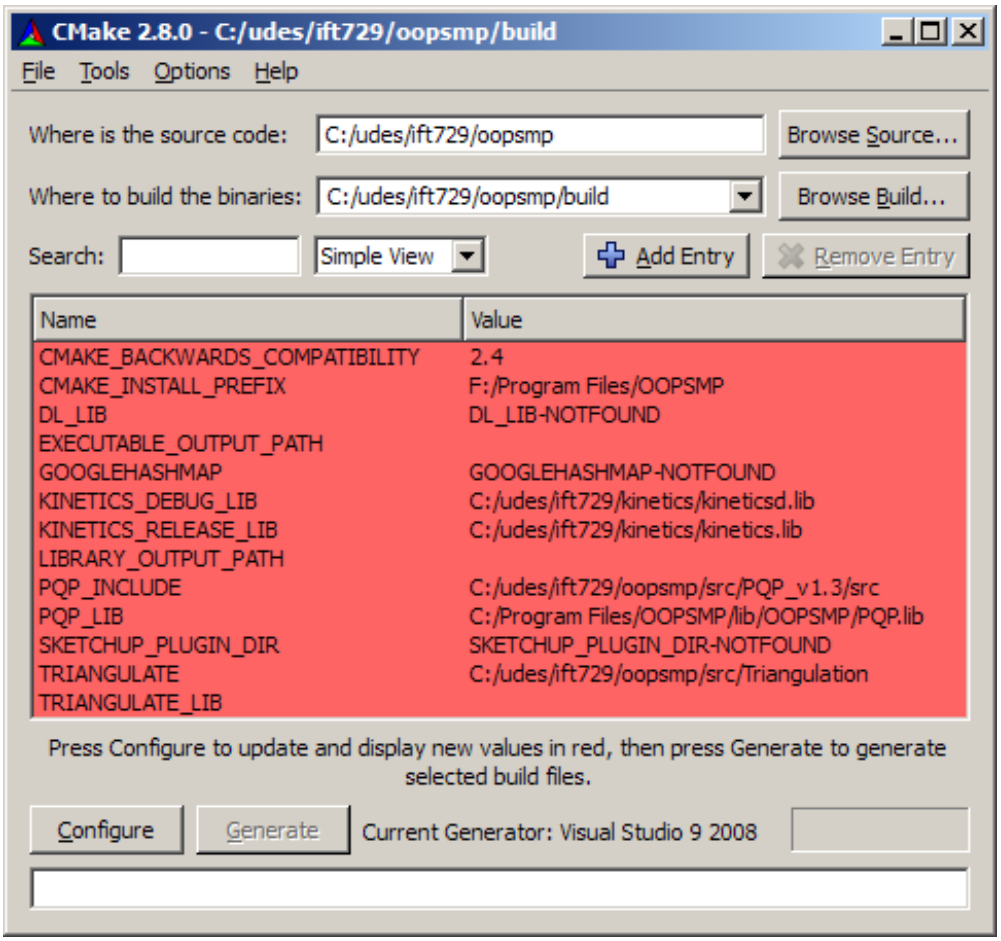



FIG. A.1 – Configuration de CMake.

Une fois les fichiers projet configurés et générés (boutons *Configure* et *Generate* dans CMake, respectivement), il faut ouvrir le fichier `build\OOPSMP.sln` (dans l'arborescence du code source du projet) dans Visual Studio .NET. Enfin, le code source peut être compilé en sélectionnant l'option *Build* du projet `ALL_BUILD`. Ce projet de la solution se chargera de compiler tous les projets dans l'ordre, selon les dépendances. La configuration *Release* permettra d'obtenir de bien meilleures performances, bien entendu. Finalement, OOPSMP doit être installé en sélectionnant l'option *Build* du projet `INSTALL`. Les binaires de OOPSMP seront copiés par défaut dans le répertoire `C:\Program Files\OOPSMP\`, dont le sous-répertoire `bin\` doit être ajouté au `PATH` de Windows. Cette étape d'installation est **nécessaire** et doit être répétée **chaque fois** que le code source est modifié et compilé.

## A.4 *Controller Chooser*

Une fois le projet compilé, elle doit être lancée à partir d'une application que nous avons créé qui sert aussi à choisir les paramètres d'exécution du programme (figure A.2). Cette application se trouve sous le répertoire `Selector\ControllerChooser.exe` dans l'arborescence du code source du projet. Une fois les paramètres choisis, OOPSMP peut être lancé en cliquant sur le bouton *Generate and run !* de l'interface.

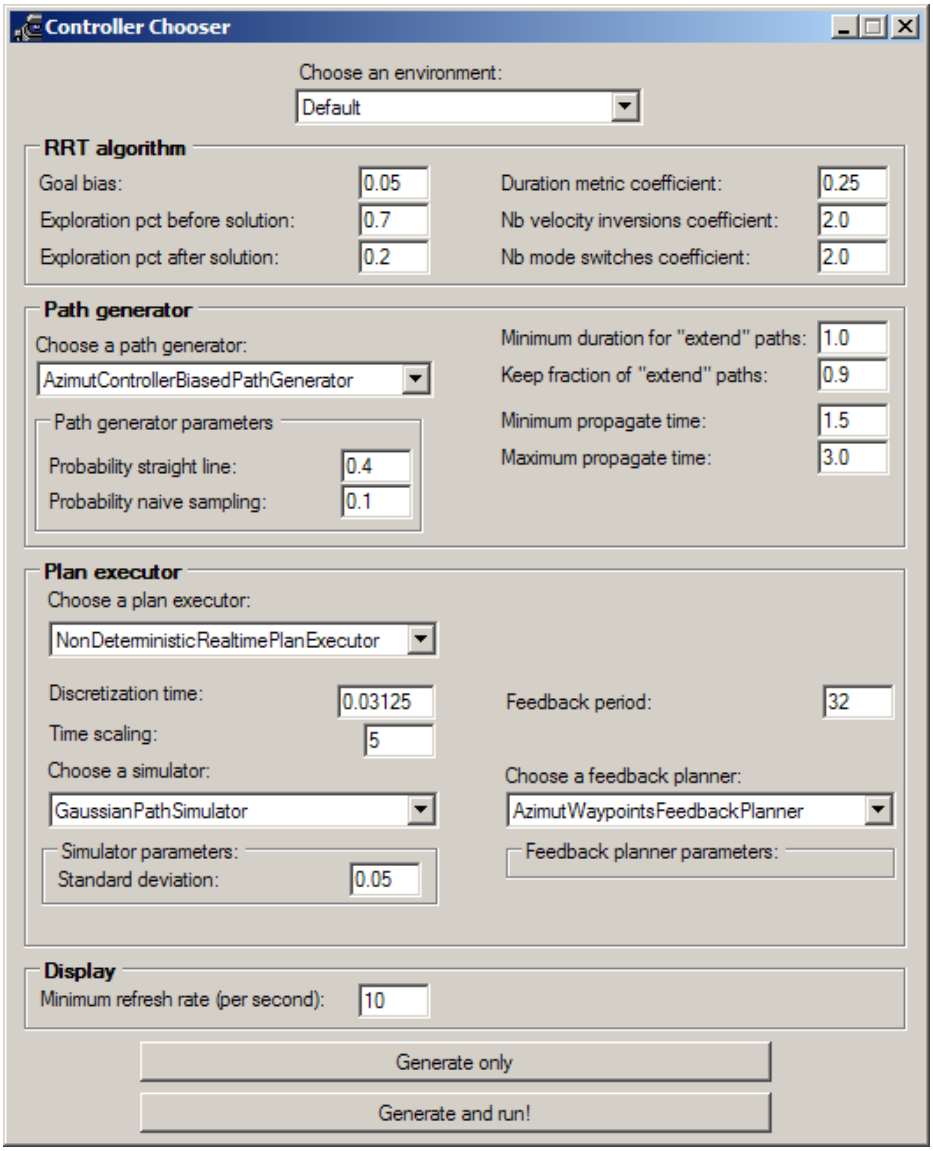



FIG. A.2 – Interface du *Controller Chooser*.

## A.5 Exécution

Une fois l'application lancée avec les paramètres choisis à partir du *Controller Chooser*, plusieurs fonctionnalités sont disponibles à partir de l'interface graphique de OOPSMP (figure A.3). Cette section détaille les procédures à suivre pour accomplir quelques tâches dignes d'intérêt.

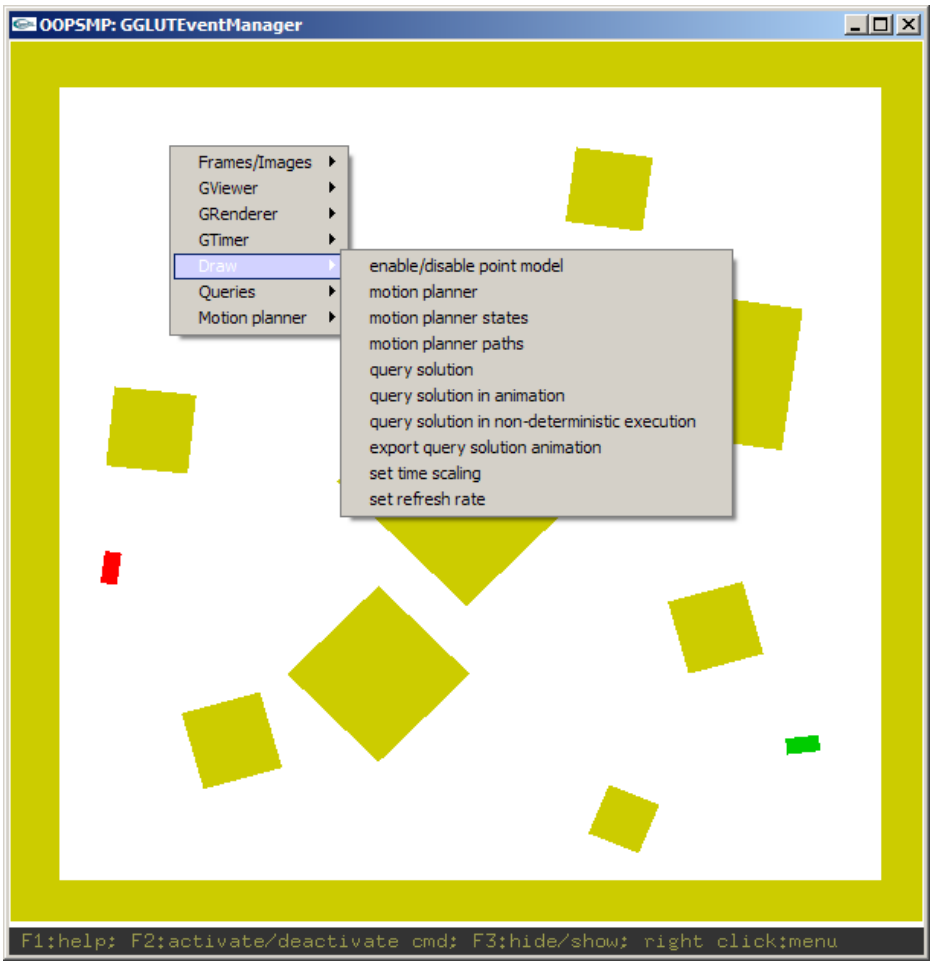



FIG. A.3 – Interface graphique de OOPSMP.

### A.5.1 Planification d'une trajectoire

Afin de procéder à la planification d'une trajectoire, il faut tout d'abord choisir une requête, constituée d'une paire état initial—état final. Comme c'est le cas pour la plupart des fonctionnalités de OOPSMP, la sélection d'une requête peut se faire via le menu contextuel (ouvert avec

un clic droit sur l'interface) ou à l'aide d'un raccourci clavier. Avec le menu, les commandes *Queries → next* et *Queries → previous* passent, respectivement, à la prochaine et à la précédente requête. Les raccourcis-clavier sont, respectivement, `n` et `p`.

Ensuite, la planification d'une trajectoire se fait en deux étapes. Il faut d'abord effectuer un prétraitement grâce à la commande *Motion planner → preprocess* (`r`) puis tenter de résoudre le problème avec la commande *Motion planner → solve* (`s`). Il est possible d'effectuer les deux opérations d'un seul coup avec la commande *Motion planner → one-click solve* (`q`), qui ne fait qu'exécuter les deux commandes précédentes successivement. Ce processus doit être répété jusqu'à ce que la console affiche le message *"query is solved"*, indiquant qu'une solution pour la requête a été trouvée.

### A.5.2 Exécution d'une trajectoire

Il existe trois façons différentes d'afficher une trajectoire qui a été calculée. La première ne fait que dessiner tous les états que prendra successivement l'objet mobile simulé, tout au long de sa trajectoire. Cette option est accessible via le menu par la commande *Draw → query solution*. La deuxième méthode, activée par la commande *Draw → query solution in animation*, affiche, dans une animation, l'exécution de la trajectoire par l'objet. La ligne noire indique la trajectoire qui a été planifiée et l'objet bleu est l'état actuel du corps simulé. Le troisième et dernier mode est celui qui nous intéresse le plus dans le cadre de ce projet. Il s'agit de la commande *Draw → query solution in non-deterministic execution*. Comme le nom l'indique, c'est cette commande qui permet d'utiliser la triade de classes détaillées dans la section 2 pour exécuter la trajectoire de façon non-déterministe. La ligne noire illustre, comme pour la deuxième méthode, la trajectoire planifiée et qui devrait être suivie aussi près que possible. L'object bleu représente l'état actuel du corps simulé et, si le `WaypointsFeedbackPlanner` a été sélectionné comme module de suivi de trajectoires, le polygone magenta indique le point de cheminement suivant vers lequel l'objet mobile devrait se diriger.

## A.6 Désinstallation

Afin de désinstaller le projet, il ne suffit que de désinstaller les applications et les bibliothèques installés dans la section A.2. Il faut aussi effacer le répertoire où se trouve le code source du projet ainsi que le répertoire où OOPSMP s'installe (`C:\Program Files\OOPSMP\` par défaut). Le sous-répertoire `bin\` peut ensuite être enlevé de la variable d'environnement `PATH` de Windows en toute sécurité.

# B Génération de trajectoires pour une voiture simple

## B.1 Hypothèses

Pour le modèle simple de la voiture, certaines hypothèses sont assumées pour simplifier le problème [3]. Tout d'abord, l'état du véhicule à un moment $t$ donné est donné par le triplet $(x, y, \theta)$ qui décrit sa position ainsi que sa direction. Un contrôle appliqué à un état de la voiture pour une certaine durée est défini par le couple $(v, \varphi)$ qui donne, respectivement, la vélocité du véhicule et l'orientation de ses roues avant. Ceci implique donc que la voiture a une accélération infinie, c'est-à-dire qu'elle peut effectuer des changements de vitesse instantanés. De plus, il est assumé que l'angle de rotation des roues peut aussi être changé de façon instantanée.

## B.2 Modèle mathématique

La voiture possède plusieurs paramètres intrinsèques (par exemple, sa vélocité minimale et maximale), dont son empattement $L$, soit la distance entre l'essieu avant et l'essieu arrière. Tel que mentionné dans la section précédente, l'état du véhicule est représenté par un triplet $(x, y, \theta)$. Il est important de noter que la position $(x, y)$ correspond au milieu de l'essieu arrière, et non au centre du véhicule, comme c'est le cas pour d'autres robots mobiles. La figure B.1 illustre l'ensemble des paramètres nécessaires pour générer une trajectoire pour le modèle de la voiture simple.

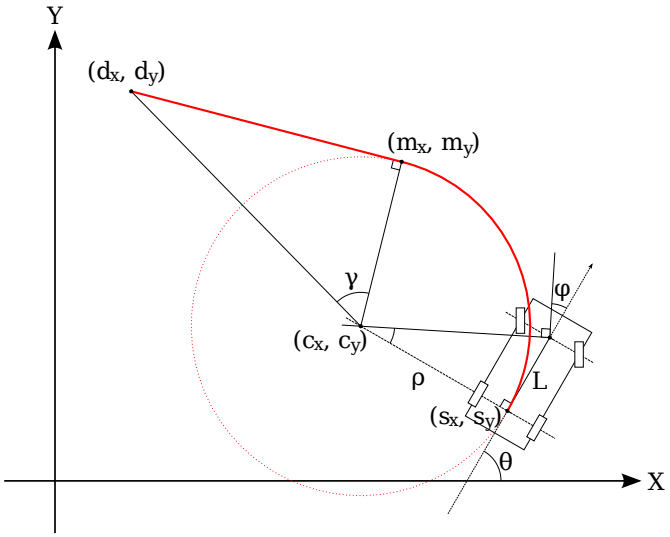



Fig. B.1 – Modèle mathématique d'une voiture simple.

L'orientation des roues avant de la voiture ($\varphi$) et l'empattement permettent de définir un cercle de centre $(c_x, c_y)$ et de rayon $|\rho|$ autour duquel la voiture tourne. La valeur $\rho$ est calculé de cette façon :

$$\rho = \frac{L}{\tan \varphi}$$

$\rho$ aura une valeur positive si l'orientation des roues $\varphi$ est positive et vice versa (un angle d'orientation des roues de 0 *rad* correspond à la direction $\theta$ de la voiture). Le signe de $\rho$ indique que le cercle est situé à gauche de la voiture s'il est positif et à droite s'il est négatif.

Le centre du cercle est calculé en fonction de $\rho$ ainsi que de l'état actuel de la voiture $(s_x, s_y, \theta)$ :

$$\begin{aligned} c_x &= s_x + \rho \cdot \cos\left(\theta + \frac{\pi}{2}\right) \\ c_y &= s_y + \rho \cdot \sin\left(\theta + \frac{\pi}{2}\right) \end{aligned} \tag{B.1}$$

L'angle de l'arc de cercle parcouru par la voiture en appliquant le contrôle $(v, \varphi)$ durant un certain temps $t$ est calculé de la façon suivante :

$$\alpha = \frac{v \cdot t}{\rho} + \theta - \frac{\pi}{2}$$

L'état final $(d_x, d_y, d_\theta)$ obtenu en appliquant $(v, \varphi)$ durant un temps $t$ peut enfin être calculé :

$$\begin{aligned} d_x &= c_x + \rho \cdot \cos \alpha \\ d_y &= c_y + \rho \cdot \sin \alpha \\ d_\theta &= \alpha + \frac{\pi}{2} \end{aligned}$$

Ces équations permettent donc de calculer un état final, étant donné un état initial, un contrôle et une durée, en temps constant, plutôt que de devoir recourir à une technique d'intégration numérique pouvant prendre un temps d'exécution arbitrairement long.

## B.3 Génération d'une trajectoire

Une trajectoire pour la voiture simple est composée de deux sous-trajectoires : un parcours sur un arc de cercle et une trajectoire en ligne droite (voir la ligne rouge grasse sur la figure B.1).

La première sous-trajectoire consiste à réorienter le véhicule en direction de son objectif $(d_x, d_y)$ en le faisant tourner autour d'un cercle de rayon minimal (l'orientation des roues avant est donc maximal). Le deuxième segment de la trajectoire sera tangent au cercle ; il convient donc de calculer le point intermédiaire $(m_x, m_y)$ tel que le segment définit par les points $(m_x, m_y)$ et $(d_x, d_y)$ est tangent au cercle.

Le calcul du point $(m_x, m_y)$ s'effectue à partir du centre du cercle (trouvé dans l'équation (B.1)), de l'angle $\beta$ du segment formé par les points $(c_x, c_y)$ et $(d_x, d_y)$, ainsi que de l'angle $\gamma$ (voir la figure B.1). $\gamma$ est calculé à partir de $\rho$ et de la longueur du segment $\overline{CD}$ :

$$\gamma = \cos^{-1}\left(\frac{|\rho|}{|\overline{CD}|}\right) \tag{B.2}$$

L'état de la voiture au point $(m_x, m_y)$ peut enfin être calculé à partir des résultats trouvés dans les équations (B.1) et (B.2) :

$$\begin{aligned} m_x &= c_x + |\rho| \cdot \cos\left(\beta - \text{sign}\left(\varphi\right) \cdot \gamma\right) \\ m_y &= c_y + |\rho| \cdot \sin\left(\beta - \text{sign}\left(\varphi\right) \cdot \gamma\right) \\ m_\theta &= \beta + \text{sign}\left(\varphi\right) \cdot \left(\frac{\pi}{2} - \gamma\right) \end{aligned}$$

Le calcul de la sous-trajectoire en ligne droite entre les points $(m_x, m_y)$ et $(d_x, d_y)$ est trivial et n'est donc pas détaillé dans le présent document.

# Bibliographie

[1] Malik Ghallab, Dana Nau et Paolo Traverso : *Automated Planning : Theory and Practice*. Morgan Kaufmann, 2004.

[2] Jean-Claude Latombe : *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[3] Stephen M. LaValle : *Planning Algorithms*. Cambridge University Press, 2006.

[4] Erion Plaku, Kostas E. Bekris et Lydia E. Kavraki : OOPS for motion planning : An online open-source programming system. Dans *IEEE International Conference on Robotics and Automation*, pages 3711–3716, 2007.

[5] Anthony Stentz : The focussed D* algorithm for real-time replanning. Dans *International Joint Conference on Artificial Intelligence*, pages 1652–1659, 1995.